Limitations of Automating Model Migration in Response to Metamodel Adaptation

Markus Herrmannsdoerfer and Daniel Ratiu

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
{herrmama, ratiu}@in.tum.de

Abstract. In consequence of changing requirements and technological progress, modeling languages are subject to change. When their metamodels are adapted to reflect those changes, existing models might become invalid. Manually migrating the models to the adapted metamodel is tedious. To substantially reduce effort, a number of approaches have been proposed to fully automate model migration. However, modeling language evolution occasionally leads to metamodel changes for which the migration of models inherently cannot be fully automated. In these cases, the migration of models requires information which is not available in the model. If such changes are ignored or circumvented, they may lead to language erosion. In this paper, we formally characterize metamodel adaptations in terms of the effort needed for model migration. We focus on the problem of metamodel changes that prevent the automatic migration of models. We outline different possibilities to systematically cope with these kinds of metamodel changes.

1 Introduction

Even though often neglected, a modeling language is subject to change like any other software artifact [1]. This holds for both general-purpose and domain-specific modeling languages. For instance, UML [2] – a general purpose modeling language – already has a rich evolution history, although it is relatively young. Domain-specific modeling languages – like e. g. AUTOSAR [3] for the specification of automotive software architectures – are even more prone to change, as they have to be adapted, whenever their domain changes due to technological progress or evolving requirements.

A modeling language is evolved by *adapting* its metamodel to the evolved requirements. Due to metamodel adaptation, existing models may no longer conform to the adapted metamodel. These models have to be *migrated* so that they can be used with the new version of the modeling language. Throughout the paper, the combination of metamodel adaptation and reconciling model migration is referred to as *coupled evolution*. Manually migrating existing models to the adapted metamodel is tedious and error-prone. Adequate tool support is thus required to reduce the effort for model migration.

To determine requirements for adequate tool support, we *reverse engineered* the coupled evolution of a number of real-world modeling languages [4, 5]. More specifically, we classified the performed metamodel changes according to the automatability of the associated model migration. *Metamodel-only* changes do not require the migration of models, whereas *coupled changes* do. *Metamodel-independent* coupled changes do not depend on a specific metamodel. *Metamodel-specific* coupled changes are so specific to a certain metamodel that the migration cannot be reused across metamodels. *Model-specific* coupled changes require information from the modeler during migration, and thus the migration cannot be specified in a model-independent way. As no changes were classified as model-specific, we decided to ignore them for the first version of our tool support called COPE. A great threat to the study's validity is that the developers might have feared and thus avoided model-specific coupled changes.

With adequate tool support at our hands, we started using COPE to *forward engineer* the coupled evolution of a number of modeling languages. From our experience with adapting several metamodels, we learned that sometimes changes requested by the users of the modeling language require a metamodel adaptation that prevents the automation of the model migration. Using our terminology presented above, we would say that these cases require a *model-specific coupled evolution*. Implementing such a change would invalidate the existing models, which have to be migrated manually. However, manually migrating a potentially unknown number of models imposes a heavy burden on the language users. Consequently, the language developers are tempted to avoid model-specific coupled evolution by adapting the metamodel in a way that the model migration does not require information from the users. However, not being able to implement such changes limits modeling language evolution, and threatens the simplicity and quality of the metamodel. Unfortunately, existing approaches are not able to cope with model-specific coupled changes.

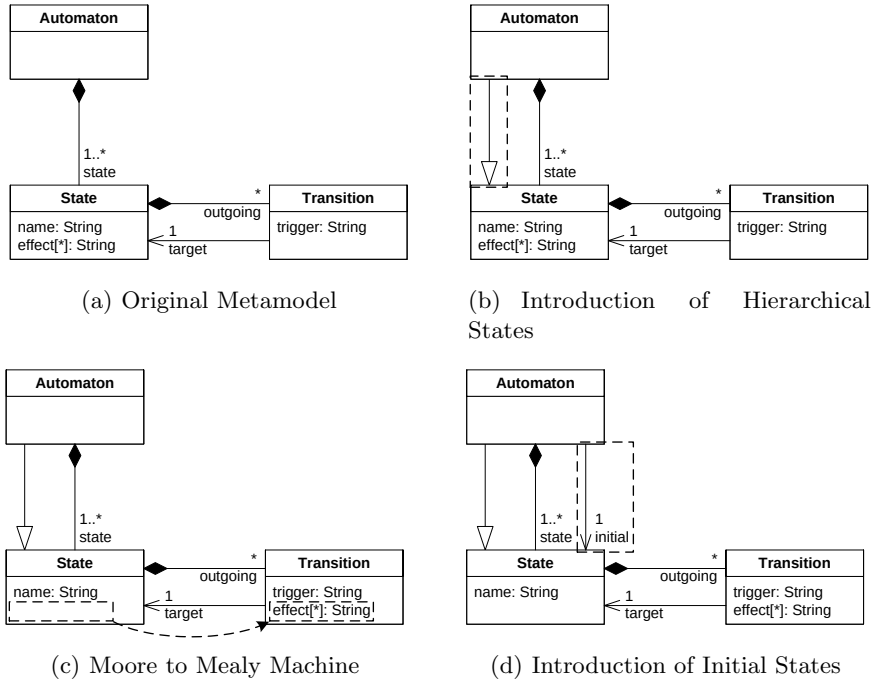
In this paper, we provide a formal framework to characterize metamodel adaptations in terms of the efforts needed for model migration. We focus on the problem of metamodel changes that prevent the automatic migration of models. We outline different possibilities to systematically cope with these kinds of metamodel changes.

Outline. In Sec. 2, we provide a number of examples to distinguish model-specific coupled changes from other kinds of changes. In Sec. 3, we present the formal framework to generally characterize model-specific coupled evolution. In Sec. 4, we outline a number of possible solutions to cope with model-specific coupled evolution In Sec. 5, we analyze related work with respect to their support for model-specific coupled evolution, before we conclude in Sec. 6.

2 Motivating Example

We chose a simple modeling language to specify automata as a running example. To show different levels of automating a model migration, we present a number

of adaptations to the language's metamodel. Fig. 2 depicts the different versions of the metamodel as UML class diagrams[1].



(a) Original Metamodel

(b) Introduction of Hierarchical States

(c) Moore to Mealy Machine

(d) Introduction of Initial States

Fig. 1. Metamodel Adaptation

Fig. 1(a) depicts the original version of the metamodel. An Automaton defines a number of states each of which is identified by a name. A State has a number of outgoing transitions each of which is activated by a trigger. When a Transition is activated, the control transitions to a target state. When the target state is entered, a sequence of actions is performed as effect. Strings are used for state names and to denote triggers and effects.

As is depicted in Fig. 1(b), we introduce hierarchical states to our modeling language to be able to structure complex automata. In the metamodel, we thus make Automaton a subclass of State. Now, a State can also be an Automaton which can again be decomposed into a number of states, and so on. Existing models are not affected by this metamodel adaptation, as they are still valid and have the same meaning. As a consequence, they do no have to be migrated, and the adaptation basically is a conservative extension.

[1] For better overview, metamodel adaptations are highlighted by dashed boxes in the figure.

As is depicted in Fig. 1(c), we change the modeling language from a Moore to a Mealy automaton to ease the specification of effects. In Moore machines, the effect of the automaton only depends on the current state. In contrast, the effect of the automaton depends also on the trigger in Mealy machines. In the metamodel, we thus move the attribute effect from State to Transition. In response, models no longer conform to the adapted metamodel, as an effect is no longer allowed for a state. To migrate these models, the effect of each state has to be moved to all its incoming transitions. Note that this model migration can be fully automated by the well-known algorithm to transform Moore automata to Mealy automata.

As is depicted in Fig.1(d), we introduce initial states to refine the modeling language. In the metamodel, the reference initial is introduced to denote the initial state within an automaton. As this reference is mandatory, the model needs to be migrated to add the missing information. However, the initial states cannot be unambiguously inferred from the information already available in the model, and default values cannot be provided either. As a consequence, this model migration cannot be automated, as information is required from the developer of the model during migration.

One solution would be to avoid the model-specific change. For example, we could make the new reference initial optional instead of mandatory in order to not invalidate existing models. However, our metamodel would not be as robust as we want it to be, and our tools would have to cope with missing initial states in an ad-hoc manner (each tool can interpret the initial states differently) and this would lead to ambiguities. As a result, this reduces the overall quality and simplicity of the modeling language and makes models fragile. To prevent such language erosion, we need to be able to support model-specific coupled evolution.

3 Characterizing Model-specific Migration

Language definition. Before we can characterize evolution, we have to formally define our understanding of a modeling language. A modeling language is completely specified through its abstract syntax and semantics [6].

Definition 1 (Modeling language). *A modeling language $\mathcal{L} = (\mathcal{M}, \mathcal{S})$ is a tuple of abstract syntax $\mathcal{M}$ and semantics $\mathcal{S}$. The abstract syntax $\mathcal{M} = \{m_1, m_2, \ldots\}$ defines the (possibly infinite) set of all models that are syntactically correct. The semantics $\mathcal{S} : \mathcal{M} \rightarrow \mathcal{SD}$ is defined by a mapping from the models $\mathcal{M}$ to a semantic domain $\mathcal{SD}$.*

A model is syntactically valid if it is built from the constructs and fulfills the constraints defined by the metamodel. For the sake of simplicity, we omit the formalization of the relationship between model and metamodel here. Two models $m_1, m_2 \in \mathcal{M}$ are syntactically equivalent if and only if $m_1 = m_2$. In practice, usually only a small subset $\mathcal{M}_{built} \subset \mathcal{M}$ of the possible models are actually built. Two models $m_1, m_2 \in \mathcal{M}$ are semantically equivalent, i. e. $\mathcal{S}(m_1) \equiv \mathcal{S}(m_2)$, if the interpretation returns equivalent objects from the semantic domain.

Language evolution. A language evolution is usually reflected by the adaptation of the metamodel and/or in the adaptation of the semantic mappings. In the following, we talk about two language versions $\mathcal{L}_1 = (\mathcal{M}_1, \mathcal{S}_1)$ and $\mathcal{L}_2 = (\mathcal{M}_2, \mathcal{S}_2)$, where $\mathcal{L}_1$ evolved to $\mathcal{L}_2$. If the language evolution is an extension, we do not need to migrate existing models.

Definition 2 (Conservative extension). *$\mathcal{L}_2$ is a conservative extension of $\mathcal{L}_1$ if and only if $\mathcal{M}_1 \subset \mathcal{M}_2$ and $\mathcal{S}_1(m) \equiv \mathcal{S}_2(m)$ for all $m \in \mathcal{M}_1$.*

A conservative extension thus does not syntactically invalidate existing models, and preserves their meaning. An extension can occur when new constructs are added to the metamodel, when existing constructs are extended, or when constraints are removed or weakened. For instance, in Fig. 1(b), we extended the modeling language with hierarchical states. Existing flat automata are still valid with respect to the adapted metamodel, and have the same meaning.

Definition 3 (Restriction). *$\mathcal{L}_2$ is a restriction of $\mathcal{L}_1$ if and only if $\mathcal{M}_2 \subset \mathcal{M}_1$.*

A restriction syntactically invalidates a number of models, namely $\mathcal{M}_1 \backslash \mathcal{M}_2$, which have to be migrated. A restriction can occur when constructs are removed from the metamodel, when existing constructs are restricted, or when constraints are added or strengthened. For instance, evolving from Fig. 1(b) back to Fig. 1(a), is a restriction, as automata with hierarchy are no longer valid and thus have to be flattened.

In general, a new version of the language exhibits both restrictions and extensions which we call a variation.

Definition 4 (Variation). *$\mathcal{L}_2$ is a variation of $\mathcal{L}_1$ if $\mathcal{M}_2 \setminus \mathcal{M}_1 \neq \emptyset \ \wedge \ \mathcal{M}_1 \setminus \mathcal{M}_2 \neq \emptyset$.*

A variation invalidates outdated models, namely $\mathcal{M}_o = \mathcal{M}_1 \backslash \mathcal{M}_2$, which have to be migrated. In practice, only those outdated models need to be migrated that are actually built, i. e. $\mathcal{M}_{built} \cap \mathcal{M}_o$. For instance, in Fig. 1(c), we change the language from a Moore to Mealy automata. All models that define effects for their states are no longer valid, but we now allow new models that define effects for their transitions. We have to find a mapping that migrates the outdated models to the new language version, and at the same time preserves the semantics of the models.

Model migration. As we have seen before, models may have to be migrated to preserve them in response to language evolution. A model migration is defined by a migration function which maps the models from the old language $\mathcal{L}_1$ to models of the new language $\mathcal{L}_2$.

Definition 5 (Migration function). *A migration function is a function $\mu : \mathcal{M}_1 \to \mathcal{M}_2$ which maps each model $m \in \mathcal{M}_1$ to a model $\mu(m) \in \mathcal{M}_2$ such that $\mathcal{S}_1(m) \equiv \mathcal{S}_2(\mu(m))$.*

Note that we explicitly require that the migration function preserves the semantics of all models. Otherwise, information is lost during migration which needs to be avoided. For instance, the transition from a Moore to a Mealy automaton (see Fig. 1(c)) requires a migration function that maps all models with effects in states to models with effects in transitions. These models have to be mapped in a way that the effects are moved for each state to all incoming transitions.

However, there are cases in which the mapping cannot be performed without information which is not available in the model.

Definition 6 (Model-specific migration). *A migration μ is model-specific if and only if a model $m_1 \in \mathcal{M}_1$ exists for which $|\mathcal{E}(m_1)| > 1$, where $\mathcal{E}(m_1) = \{m_2 \in \mathcal{M}_2 \mid \mathcal{S}_1(m_1) \equiv \mathcal{S}_2(m_2)\}$.*

A migration has to be model-specific if several models in $\mathcal{M}_2$ exist which are semantically equivalent to a model in $\mathcal{M}_1$. In this case, additional information is necessary during migration to choose one of these semantically equivalent models. The set for which the migration has to be model-specific is $\mathcal{M}_{ms} = \{m_1 \in \mathcal{M}_1 \mid |\mathcal{E}(m_1)| > 1\}$. Again, model-specific migration is only required for models from $\mathcal{M}_{built} \cap \mathcal{M}_{ms}$.

For instance, in Fig. 1(d), we introduced mandatory initial states which leads to a model-specific migration. A number of models with different choices for initial states are semantically equivalent to each model without initial states. Model-specific information is required to choose the one that is intended by the developer of the model.

A migration is thus model-independent if $|\mathcal{E}(m_1)| = 1$ for all $m_1 \in \mathcal{M}_1$. For instance, for the transition from a Moore to a Mealy automaton (see Fig. 1(c)), there is exactly one model in $\mathcal{M}_2$ that is semantically equivalent to a model in $\mathcal{M}_1$. Consequently, no additional information is required during migration, and we can thus specify an algorithm that automatically performs the migration.

4 Coping with Model-specific Migration

We outline a number of possible solutions to cope with model-specific coupled evolution. The solutions take advantage of particular situations which are however encountered very often in practice.

Effort analysis. In practice, only a small subset of all possible models $\mathcal{M}$ are actually built. Only the existing models need to be migrated. In many practical situations, (e. g. for highly domain-specific languages) the language developers and users are quite close to each other (e. g. in the same company). In these situations, it is often the case that the entire set of the existing models is known to the language developers. For instance, whenever the language is used only in-house all models are contained in a central repository.

In case when all models are known, language developers can make informed language improvements also with respect to the effort needed for model migration. Based on the existing models, they can assess the manual migration effort

required after metamodel adaptation. For instance, the effort needed to introduce initial states is proportional to the number of automata in the existing models. They can decide whether an improvement in the language is worth making given the amount of manual work necessary to migrate the existent models. However, in a lot of cases, language developers and users are decoupled which makes this approach infeasible.

Interactive migration. One possibility to support model-specific coupled evolution is to provide user interaction during migration. The migration algorithm automatically migrates the model as far as possible, and whenever it needs supplementary information, it asks the language user to provide the missing information. It can also suggest a number of alternatives from which the language user has to choose. We have extended the language provided by our approach COPE with a primitive to trigger such an interaction during migration.

Listing 1.1 shows the use of this primitive in a coupled operation that introduces initial states in the language. The metamodel adaptation creates the new reference initial which is single-valued and mandatory. During model migration, the developer of the model has to choose an initial state for each automaton. The primitive choose takes three parameters as input, namely the context element, the values to choose from and a message, and returns the chosen value.

Listing 1.1. Interactive Coupled Operation

```
// metamodel adaptation
newReference(Automaton, "initial", State, 1, 1, !CONTAINMENT)

// model migration
for(a in Automaton.allInstances) {
        a.initial = choose(a, a.state, "Choose initial state")
}
```

Figure 2 shows the dialog that is opened during model migration to let the language user make a choice. The dialog shows the current state of the model selecting the context element, the list of values to choose from, as well as the message. The developer of the model is required to choose a value from the list to determine the initial state within an automaton.

Implicit information. Many times, the language users employ different conventions (e. g. naming conventions) to capture more information than is made explicit through the metamodel. In these cases, the language user can incorporate implicit information to help automate the migration. For example, language users might have already named all initial states with a prefix "I", before the initial states were explicitly introduced in the metamodel. They can then refine the migration in a way that for each automaton, it automatically chooses the

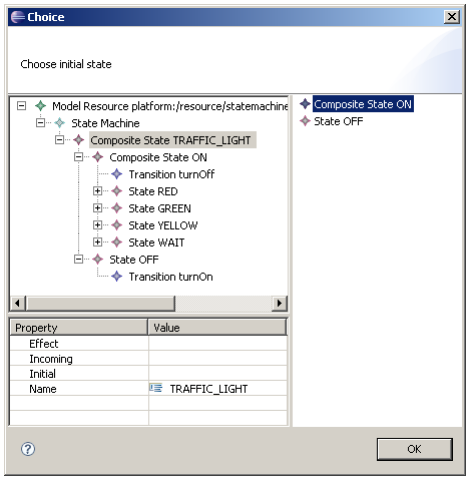


Fig. 2. UI for interactive coupled operations

marked sub state. We thus plan to introduce a means to allow the language user to upfront establish certain choices introduced by the language developer.

5 Related Work

Current approaches to reduce the effort for model migration focus on fully automating the model migration. They thus do not address model-specific coupled evolution which can inherently not be fully automated. We believe that the current language evolution approaches do not support model-specific migration, because the problem is not completely understood yet. With this paper, we thus aimed to characterize the problem, and outlined possible solutions. Related work on model migration can be subdivided into two kinds of approaches: difference-based and operation-based.

Difference-based approaches allow language developers to synthesize a model migration based on the difference between two metamodel versions. Sprinkle et al. introduce a visual graph-transformation-based language which requires to specify the model migration only for the metamodel difference [7]. However, this language does not provide constructs to cater for model-specific migrations. The following approaches further automate model migration by automatically deriving it from the difference between two metamodel versions. Gruschko et al. classify primitive metamodel changes into non-breaking, breaking resolvable and unresolvable changes [8, 9]. Based on this classification, they propose to automatically derive a migration for non-breaking and resolvable changes. Cichetti et al. go even one step further and try to detect composite changes like e. g. extract class in the difference between metamodel versions [10, 11]. Garces et al. refine this approach, present a prototype, and demonstrate its applicability in

two case studies [12]. However, the derivation approaches are not able to detect model-specific migrations in the metamodel difference.

Operation-based approaches allow language developers to incrementally transform the metamodel by means of coupled operations which also encapsulate the corresponding model migration. Although they are not as automated as difference-based approaches, they allow to capture the intended model migration already when adapting the metamodel. Hößler and Soden present a number of high-level coupled operations which automate metamodel adaptation as well as model migration [13]. Wachsmuth adopts ideas from grammar adaptation and proposes a classification of metamodel changes based on instance preservation properties [14]. Based on these preservation properties, the author defines a library of high-level coupled operations. In [5, 15] we generalize these approaches by a language which allows to specify arbitrary new high-level coupled operations. Moreover, this language can be used to attach a custom migration to a metamodel adaptation, which cannot be covered by high-level coupled operations. We demonstrate the usefulness of this approach by presenting the coupled evolution of two real-life metamodels. However, all of these approaches do not provide coupled operations which support model-specific migrations. In this paper, we thus extended our language for expressing coupled operations with a choice construct that allows the user to guide the migration.

6 Conclusion

Modeling languages evolve over time and thereby their metamodels need to be adapted. To reduce the effort for language evolution, the resulting migration of models needs to be automated. However, some metamodel changes require information during migration which is not available in the model. Consequently, these metamodel changes inherently prevent the automatic migration of models. In this paper, we presented an example of such model-specific changes and formally characterized them. Moreover, we presented a number of techniques to cope with model-specific migrations. With these techniques, language developers can make informed decisions about the effort needed for manual migration, and partially automate the manual migration by means of adequate tool support. As a consequence, language erosion can be prevented which can result from avoiding model-specific changes. It remains to be investigated how much a language suffers from systematically avoiding them. Moreover, we are interested in how often model-based changes are required in practice, when not avoiding them.

References

1. Favre, J.M.: Languages evolve too! changing the software time scale. In: 8th International Workshop on Principles of Software Evolution (IWPSE), IEEE Computer Society (2005) 33–44
2. Object Management Group: Unified Modeling Language, Superstructure, v2.1.2 (2007)

3. AUTOSAR Development Partnership: AUTOSAR Specification V3.1 (2008)
4. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of Coupled Evolution of Metamodels and Models in Practice. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: MODELS 2008. Volume 5301 of LNCS., Springer Heidelberg (2008) 645–659
5. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In: ECOOP 2009. to appear, available online at `http://cope.in.tum.de/uploads/Publications/`. (2009)
6. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of "semantics"? Computer **37**(10) (Oct. 2004) 64–72
7. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. Journal of Visual Languages and Computing **15** (2004) 291–307
8. Becker, S., Gruschko, B., Goldschmidt, T., Koziolek, H.: A Process Model and Classification Scheme for Semi-Automatic Meta-Model Evolution. In: 1st Workshop MDD, SOA und IT-Management (MSI), GI, GiTO-Verlag (2007) 35–46
9. Gruschko, B., Kolovos, D., Paige, R.: Towards synchronizing models with evolving metamodels. In: International Workshop on Model-Driven Software Evolution. (2007)
10. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In Ceballos, S., ed.: 12th International Enterprise Distributed Object Computing Conference (EDOC), IEEE Computer Society (2008)
11. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: Managing dependent changes in coupled evolution. In: ICMT2009 - International Conference on Model Transformation, Springer LNCS (July 2009)
12. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Managing model adaptation by precise detection of metamodel changes. In: Model Driven Architecture - Foundations and Applications. Volume 5562 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2009) 34–49
13. Hößler, J., Soden, M., Eichler, H.: Coevolution of Models, Metamodels and Transformations. In: Models and Human Reasoning. Wissenschaft und Technik Verlag, Berlin (2005) 129–154
14. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: ECOOP 2007. Volume 4609 of LNCS., Springer Heidelberg (2007) 600–624
15. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE: A Language for the Coupled Evolution of Metamodels and Models. In: 1st International Workshop on Model Co-Evolution and Consistency Management. (2008)